Object-Oriented Thinking Chapter

Contents

[Class Abstraction 3](#_Toc82415234)

[Class Encapsulation 4](#_Toc82415235)

[Encapsulation 4](#_Toc82415236)

[Pillars of OOP 4](#_Toc82415237)

[Class contract 4](#_Toc82415238)

[ADT 4](#_Toc82415239)

[Coupling 5](#_Toc82415240)

[Cohesion: 5](#_Toc82415241)

[Class Relationships: 6](#_Toc82415242)

[Association 6](#_Toc82415243)

[Association Represented in a UML Diagram: 6](#_Toc82415244)

[Aggregation 7](#_Toc82415245)

[Has-a 8](#_Toc82415246)

[Aggregation Represented in a UML diagram 8](#_Toc82415247)

[Composition 8](#_Toc82415248)

[Has-a 8](#_Toc82415249)

[Composition Represented in a UML diagram 9](#_Toc82415250)

[How to tell which is which? 9](#_Toc82415251)

[Generalization vs specialization 10](#_Toc82415252)

[Generalization 10](#_Toc82415253)

[Specialization 10](#_Toc82415254)

[Generalization vs Inheritance 10](#_Toc82415255)

[Processing Primitive Data Type Values as Objects 10](#_Toc82415256)

[Primitive Data Types 10](#_Toc82415257)

[Reference Data Types 11](#_Toc82415258)

[Automatic Conversion between Primitive Types and Wrapper Class Types 11](#_Toc82415259)

[Converting between data types: 11](#_Toc82415260)

[Converting Strings into numbers 11](#_Toc82415261)

[Converting numbers into Strings 12](#_Toc82415262)

[Boxing 13](#_Toc82415263)

[Unboxing 13](#_Toc82415264)

[Autoboxing & Autounboxing 13](#_Toc82415265)

[BigInteger and BigDecimal Classes 13](#_Toc82415266)

[String 13](#_Toc82415267)

[How to create String object? 14](#_Toc82415268)

[1) String Literal AKA interned String 14](#_Toc82415269)

[2) By new keyword 15](#_Toc82415270)

[There are three ways to compare String objects: 16](#_Toc82415271)

[1) By equals() method 16](#_Toc82415272)

[2) By == operator 17](#_Toc82415273)

[3) By compareTo() method: 17](#_Toc82415274)

[Replacing characters or substrings in a String 17](#_Toc82415275)

[regex 21](#_Toc82415276)

[Ways we can set up patterns: 21](#_Toc82415277)

[StingBuilder & StingBuffer Classes 23](#_Toc82415278)

[**StringBuilder Operations** 25](#_Toc82415279)

The focus of this chapter is on class design and explores the difference between procedural programming and OOP. The end goal is to see the benefits of OOP and learn how to use it effectively. This chapter also covers concepts of processing primitive data types as objects, the BigInteger and BigDecimal data types, the String class, regex (regular expressions), and the StringBuilder/StringBuffer classes.

This chapter introduces 1 (?) of the pillars of OOP: Class abstraction/Class encapsulation.

What is class abstraction? It is a concept of how we design (thus implement) an object.

Remember that we want to be able to created re-usable code, re-usable objects.

Class Abstraction:

Class abstraction separates the implementation from how the class is used.

The example I like to use is a car object. Most of us are familiar with a car, we know how to turn it on, change the gears, push pedals, and turn the steering wheel. Therefore we know how to use a car. We can get in the car and go to a destination. But, do we know the specifics (what happens under the hood) of what has to happen when we start a car? Do know the specifics of what is going on with the transmissions – and all the other parts – when we change gears? Do we know the mechanics of what is happening when we push the gas pedal or the brake pedal? Probably not, but we know when you push down on the gas, the car goes faster. This is abstraction. We know how to use an object without knowing the implementation that is occurring to make it happen.

This is what we need to do when we create objects, we need to design the object without having the user know all the specifics of what has to happen to be able to use it.

If we use an example of a Circle object, then a user should be able to create a circle and figure out the area without having to know the specific formula that is used to calculate the area.

Simply put:

* The user can use the class/object without knowing the how behind is
  + The user does not need to know what is under the hood to be able to drive the car
  + The user does not need to know the formula to calculate the area

Class Encapsulation:

When, we as designers, design an object using abstraction, then what we are also doing is encapsulating the object so that the details of the implementation are hidden from the use; thus by default, we have encapsulated the object.

Encapsulation is the details of implementation are encapsulated and are hidden from the user.

* Data encapsulation is also known as data hiding
  + Only the behavior is seen to the outside world, through the public methods
* Because using the private modifier restricts other class’s from accessing the data
  + Only accessing the data through the getters/accessors & setter/mutators
* Advantages of using encapsulation:
  + Improves maintainability, flexibility, and re-usability
  + Allows the creation of immutable objects
  + \*\*\*Does the abstraction\*\*\* - meaning the user does not know what is going on “behind the scene” but is allowed to get the job done

# Pillars of OOP

The four pillars of OOP are: encapsulation, abstraction, inheritance, and polymorphism

Or – 3 pillars: encapsulation/abstraction, inheritance, and polymorphism, as some people say that encapsulation & abstraction are 2-sides of the same coin.

Class contract:

A class contract is the collection of methods and fields that are accessible from outside the class, together with the description of how these members behave



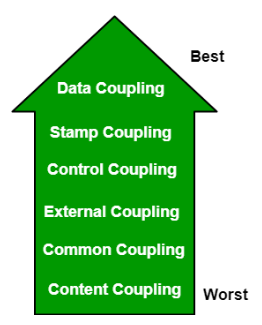
Keep in mind from here on out, that you want to separate the development of a class from the use of a class.

ADT: Abstract Data Type: you should be defining classes/objects using class abstraction & class encapsulation

Coupling:

Coupling is a concept in software design that is defined as the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules. Coupling is usually contrasted with cohesion.

* OOP paradigm couples data and methods together into object/class.
* Classes provide more flexibility and modularity for building ***reusable*** software.
* When you need something that holds data that should be tightly coupled – create an object; thus the data is now tightly coupled together



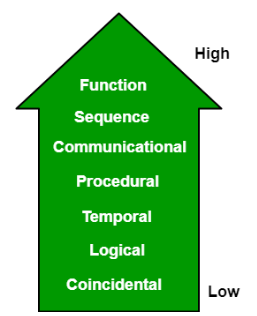
(<https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/>)

# Cohesion:

In computer programming, cohesion refers to the degree to which the elements inside a module belong together. In one sense, it is a measure of the strength of relationship between the methods and data of a class and some unifying purpose or concept served by that class. In another sense, it is a measure of the strength of relationship between the class's methods and data themselves.

Cohesion is an ordinal type of measurement and is usually described as “high cohesion” or “low cohesion”. Modules with high cohesion tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

Cohesion is often contrasted with coupling, a different concept. High cohesion often correlates with loose coupling, and vice versa.



(<https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/>)

# Class Relationships:

To design a class, we need to understand relationships that objects/classes can have.

The class relationships that we will be learning are:

* Association
* Aggregation
* Composition
* Inheritance (will be covered in the next chapter)

# Association

An associated relationship is said to be a binary relationship; it is a relationship that describes the activity between 2 classes.

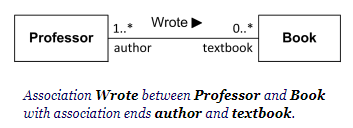
* Basically provides information of how many & direction of interaction between the objects.

Definition of association: An organization of people with a common purpose and having a formal structure

## Association Represented in a UML Diagram:

* Solid line
* Multiplicity: m..n the number of objects between m and n inclusively
* Name of the association can be shown somewhere near the middle of the association line but not too close to any of the ends of the line. Each end of the line could be decorated with the name of the association end.

Examples:



A single student can associate with multiple teachers:

Association multiplicity example 1

The example indicates that every Instructor has one or more Students:

Association multiplicity example 2

We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.

Association multiplicity example 3

# Aggregation

An aggregation relationship is a special form of association that represents an ownership relationship between two objects.

Both objects can exist on their own – one does not depend on the other to exist.

If we look at the definition of aggregation:

is a collection, or the gathering of things together

a group or mass of distinct or varied things

constituting or amounting to a whole – total

* Aggregation comes from the Latin *ad*, meaning to, and *gregare*, meaning herd.
* Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.
* It's important to note that the aggregation link doesn't state in any way that Class A owns Class B nor that there's a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link is usually used to stress the point that Class A instance is not the exclusive container of Class B instance, as in fact the same Class B instance has another container/s.

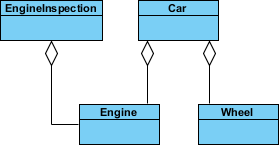
## Has-a

Aggregation is *has-a* relationship.

An object can be owned by several other aggregating objects.

Aggregation Represented in a UML diagram**:**

* Aggregation rules of:
  + Solid line with an open diamond
  + Multiplicity: m..n the number of objects between m and n inclusively



An aggregation relationship is usually represented as a data field in the aggregating class.

Aggregating object = owner

Aggregated object = subject

# Composition

A composition relationship is a special form of aggregation, a *has-a* relationship

## Has-a

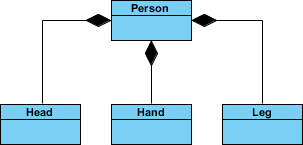
Composition is *has-a* relationship where the object is ***exclusively owned by*** an aggregating object (the owner of the object) – in other words owned by 1 object

* Composition implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.
* use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also deleted as a result

Definition: The nature of something's ingredients; the way in which a whole or mixture is made up

## Composition Represented in a UML diagram

* Composition rules of:
  + Solid line with a closed diamond
  + Multiplicity: m..n the number of objects between m and n inclusively



# How to tell which is which?

Consider the differences and similarities between the classes of the following objects: pets, cats, tails, and owners.

We see the following relationships:

* owners feed pets, pets please owners (association)
* a tail is a part of both dogs and cats (aggregation / composition)
* a cat is a kind of pet (inheritance / generalization)

The figure below shows the three types of association connectors: association, aggregation and composition.

UML association

UML aggregation

UML composition

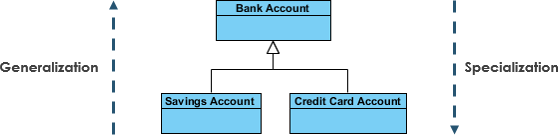
To sum it up, *association* is a very generic term used to represent when on class used the functionalities provided by another class. We say it's a composition if one parent class object owns another child class object and that child class object cannot meaningfully exist without the parent class object. If it can exist, then it is called Aggregation.

# Generalization vs specialization

Generalization is a mechanism for combining similar classes of objects into a single, more general class. Generalization identifies commonalities among a set of entities. The commonality may be of attributes, behavior, or both. In other words, a superclass has the most general attributes, operations, and relationships that may be shared with subclasses. A subclass may have more specialized attributes and operations.

Specialization is the reverse process of Generalization means creating new sub classes from an existing class.

For Example, a Bank Account is of two types - Savings Account and Credit Card Account. Savings Account and Credit Card Account inherit the common/ generalized properties like Account Number, Account Balance etc. from a Bank Account and also have their own specialized properties like unsettled payment etc.



## Generalization vs Inheritance

Generalization is the term that we use to denote abstraction of common properties into a base class in UML. *The UML diagram's Generalization association is also known as Inheritance.* When we implement Generalization in a programming language, it is often called Inheritance instead. Generalization and inheritance are basically the same, the terminology just differs depending on the context where it is being use

# Processing Primitive Data Type Values as Objects

Java uses primitive data types to be able to achieve a higher performance. Performance is measured by how much memory is used and how much processing.

Read: <http://www.drdobbs.com/jvm/a-modern-primitive-discussion/232601450>

Primitive Data Types**:**

That value is stored as in in the main memory

Issues of overflow and precision

For example, the Java statement, int num; , stores 0 in memory

Reference Data Types**:**

The reference/the location where it is in memory is stored in main

Issue of memory

For example, the Java statement, Integer num; , stores the refence to where null is stored in memory in its memory location

## Automatic Conversion between Primitive Types and Wrapper Class Types

A data type is a set of values and a set of operations defined on those values. The primitive data types that you have been using are supplemented in Java by extensive libraries of reference types that are tailored for a large variety of applications. The behavior of a data type in an application programming interface (API).

A primitive type can be automatically converted to an object using a wrapper class, and vice versa, depending on the context, using the API (the library). The wrapper classes provide constructors, constants, and conversion methods for manipulating various data types.

\*\*\* the wrapper classes do NOT have no-args constructors\*\*\*

\*\*\* all instances of the wrapper classes are immutable\*\*\*

## Converting between data types:

The \*Value() methods is an instance method of that specific Wrapper class under java.lang package. This method returns the value of the specified number as a(n) \*. It is inherited from the Number Class.

Examples:

// instantiates a Double with the value of 2302.009

Double dDoubleNum = new Double(2302.009);

// declares an int and assigns the value of 2302

int inum = dDoubleNum.intValue();

// instantiates an Integer with the value of 2302

Integer intClassNum = dDoubleNum.intValue();

// declares a double and assigns the value of 2302.009

double dnum = dDoubleNum.doubleValue();

Converting Strings into numbers**:**

The parseDataType(\*) and the valueOf(\*) methods, where \* can be just the String, or the String and specifying the numeric counting system (the radix argument)

* Binary = 2
* Octal = 8
* Decimal = 10 // the default if a radix argument is not specified
* Hexadecimal = 16

parseDataType(\*) is for primitive data types

valueOf(\*) is for reference data types

Examples:

String stringNumber = “123”;

int intFromString = Integer.parseInt(stringNumber);

Double dDoubleFromString = Double.valueOf(stringNumber);

Converting numbers into Strings**:**

* This method uses instance of Integer class to invoke it’s toString() method. The argument is converted and returned as a String instance. If the number is negative, the sign will be preserved.

int a = 1234;

int b = -1234;

String str1 = Integer.toString(a);

String str2 = Integer.toString(b);

System.out.println("String str1 = " + str1);

System.out.println("String str2 = " + str2);

int d = 1234;

Integer obj = new Integer(d);

String str4 = obj.toString();

System.out.println("String str4 = " + str4);

* Using the DecimalFormat.format() method, you can specify the number of decimal places and comma separator for readability.

int e = 12345;

DecimalFormat df = new DecimalFormat("#,###");

String Str5 = df.format(e);

System.out.println(Str5);

* Converting with a specific numbering system.

int i = 255;

String binaryString = Integer.toBinaryString(i);

System.out.println(binaryString);

Boxing: converting a primitive value to a wrapper object

Unboxing: converting a wrapper object to a primitive type

Autoboxing & Autounboxing a feature added to JDK 5: The compiler will automatically box a primitive value that appears in a context requiring an object and will unbox an object that appears in a context requiring a primitive value

int n1 = 100;

Integer n2 = new Integer(n1);

Using autoboxing: Integer n2 = 100;

# BigInteger and BigDecimal Classes

BigInteger allows very large integers

BigDecimal allows high-precision floating-point values – with no limit to the precision

* Could have issues with the divide method if it cannot be terminated
* has a better precision using the String argument than a number argument
* Methods: add, subtract, multiply, divide, and remainder

Examples:

BigInteger bigInt =

new BigInteger("123456789101112131415161718192021222324252627282930");

System.out.println(bigInt.add(bigInt));

// or

BigInteger biTwo = new BigInteger("2");

System.out.println(bigInt.multiply(biTwo));

BigDecimal bigDecimal = new BigDecimal("654.32198745632147852369");

# String

In Java, a String is basically an object that represents sequence of char values.

An array of characters works same as Java string. For example:

char[] ch = {'C','S','C', 'I,', '2', '3', '0','2'};

String s = new String(ch);

is same as:

String s = "CSCI2302";

The Java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.

The Java String is immutable, meaning it cannot be changed but a new instance is created. For a mutable class, you can use StringBuffer and StringBuilder class.

## How to create String object?

There are two ways to create String object:

By String literal 🡪 how we did in CSCI 1302

By new keyword 🡪 how we are learning now

### 1) String Literal AKA interned String

Java String literal is created by using double quotes. For Example:

String s = "welcome";

Each time you create a String literal, the JVM checks the string constant pool first. If the String already exists in the pool, a reference to the pooled instance is returned. If the String doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

String s1 = "Welcome";

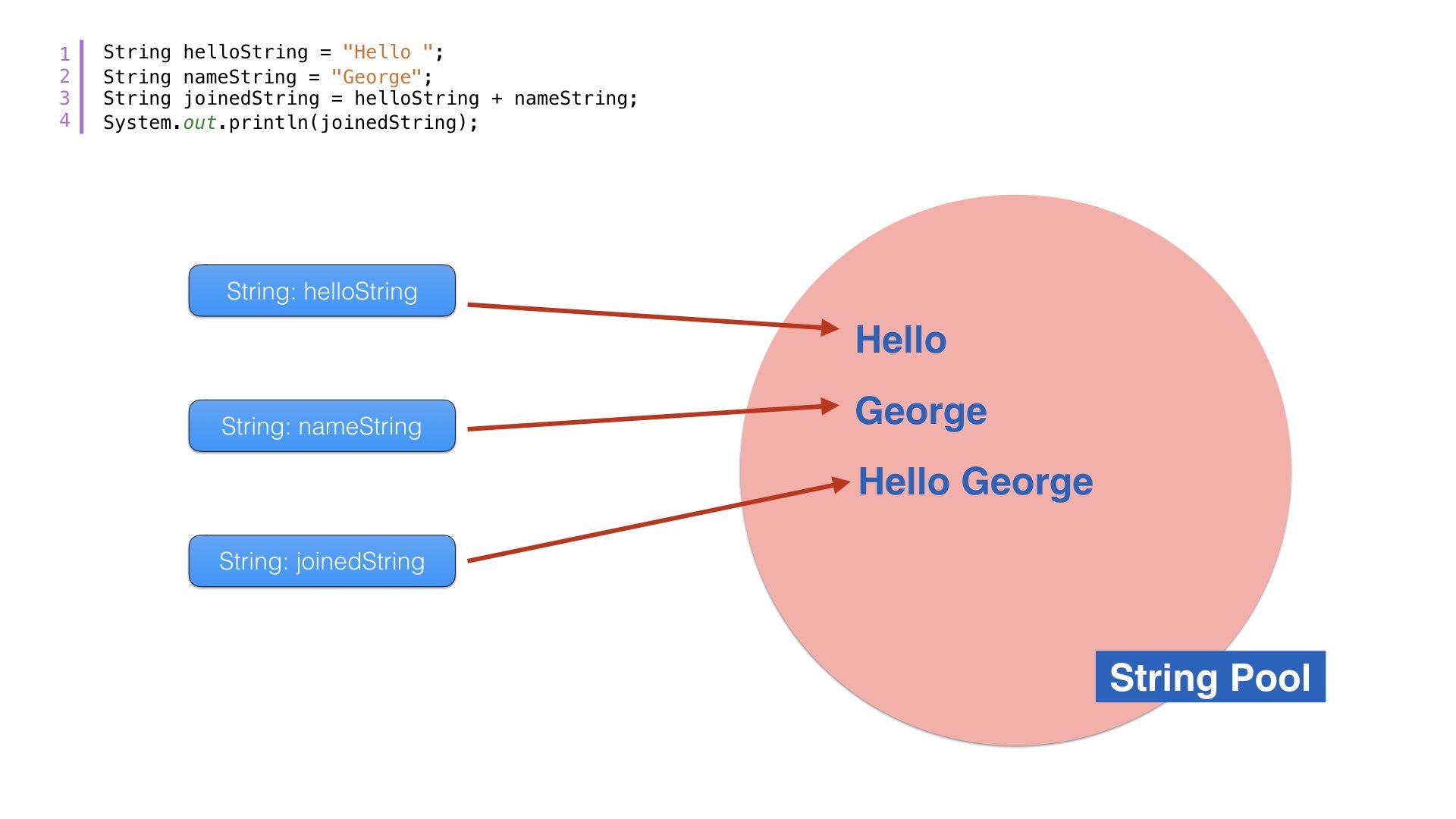
String s2 = "Welcome"; //will not create new instance

Welcome

s2

s1

In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in String constant pool, so it will create a new object. After that, it will find the String with the value "Welcome" in the pool, it will **not** create new object but will return the reference to the same instance.



Note: String objects are stored in a special memory area known as string constant pool

**Why Java uses concept of string literal?**

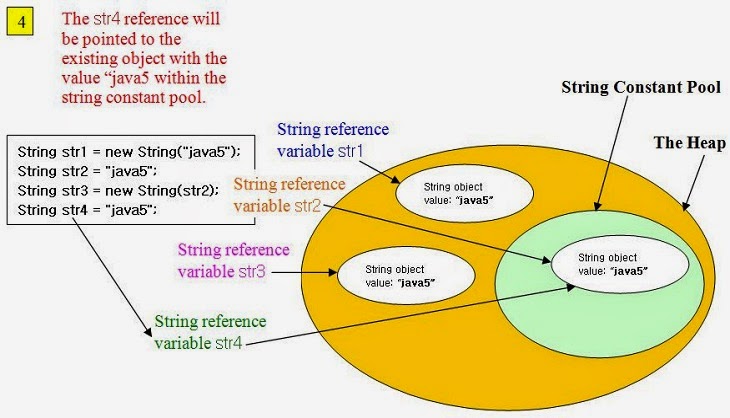
To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

### 2) By new keyword

String s = new String("Welcome");

//creates an object with a reference variable

In this case, JVM will create a new string object in normal (non pool) heap memory. The variable s will refer to the object in heap (non pool).



## There are three ways to compare String objects:

1. By equals() method
2. By = = operator
3. By compareTo() method

### 1) By equals() method

equals() method compares the original content of the String. It compares values of String for equality. String class provides two methods:

public boolean equals(Object another){} compares this string to the specified object.

public boolean equalsIgnoreCase(String another){} compares this String to another String, ignoring case.

String s1 = "CSCI2302";

String s2 = " CSCI2302";

String s3 = new String("CSCI2302");

String s4 = "CSCI1302";

System.out.println(s1.equals(s2));//true

System.out.println(s1.equals(s3));//true

System.out.println(s1.equals(s4));//false

//Example of equalsIgnoreCase(String) method

String s1 = " csci2302";

String s2 = " CSCI2302";

System.out.println(s1.equals(s2)); //false

System.out.println(s1.equalsIgnoreCase(s3)); //true

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### 2) By == operator

The == operator compares the main/method memory – NOT the values of the reference objects.

String s1 = " CSCI2302";

String s2 = " CSCI2302";

String s3 = new String("CSCI2302");

System.out.println(s1==s2);//true (because both refer to same instance)

System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### 3) By compareTo() method:

compareTo() method compares values and returns an int which tells if the values compare less than, equal, or greater than.

Suppose s1 and s2 are two String variables. If:

• s1 == s2 :0

• s1 > s2 :positive value

• s1 < s2 :negative value

String s1 = "CSCI2302";

String s2 = "CSCI2302";

String s3 = "csci2302";

System.out.println(s1.compareTo(s2));//0

System.out.println(s1.compareTo(s3));//-32(because s1 > s3)

System.out.println(s3.compareTo(s1));//32(because s3 < s1 )

## Replacing characters or substrings in a String

Methods:

|  |  |
| --- | --- |
| [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [**replace**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#replace(char,%20char))(char oldChar, char newChar)  Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar. |
| [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [**replace**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#replace(java.lang.CharSequence,%20java.lang.CharSequence))(**[CharSequence](https://docs.oracle.com/javase/7/docs/api/java/lang/CharSequence.html" \o "interface in java.lang)** target, **[CharSequence](https://docs.oracle.com/javase/7/docs/api/java/lang/CharSequence.html" \o "interface in java.lang)** replacement)  Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. |
| [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [**replaceAll**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#replaceAll(java.lang.String,%20java.lang.String))([**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) regex, [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) replacement)  Replaces each substring of this string that matches the given [**regular expression**](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#sum) with the given replacement. |
| [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [**replaceFirst**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#replaceFirst(java.lang.String,%20java.lang.String))([**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) regex, [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) replacement)  Replaces the first substring of this string that matches the given [**regular expression**](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#sum) with the given replacement. |

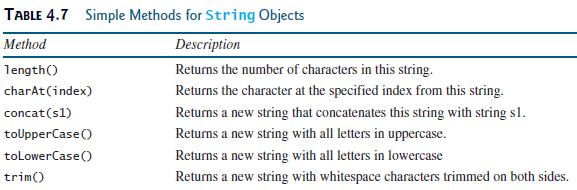
"Welcome".replace('e', 'A') returns a new string, WAlcomA.

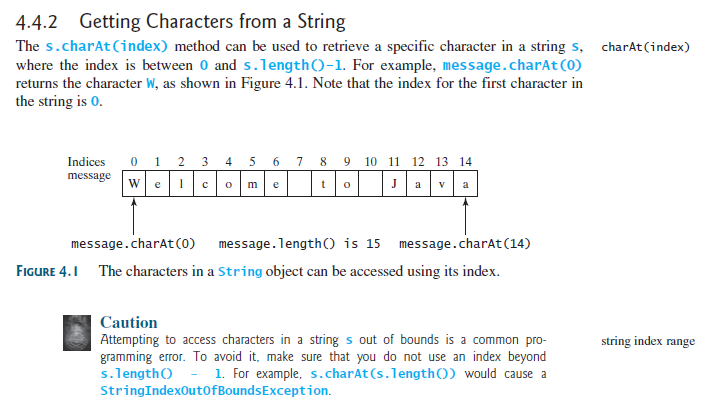
"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.

"Welcome".replace("e", "AB") returns a new string, WABlcomAB.

"Welcome".replace("el", "AB") returns a new string, WABcome.

Methods that are covered in chapter 4, Mathematical Functions, Characters, and Strings:





You can use the concat method to concatenate two strings. The statement shown below, for example, concatenates strings s1 and s2 into s3:

String s3 = s1.concat(s2);

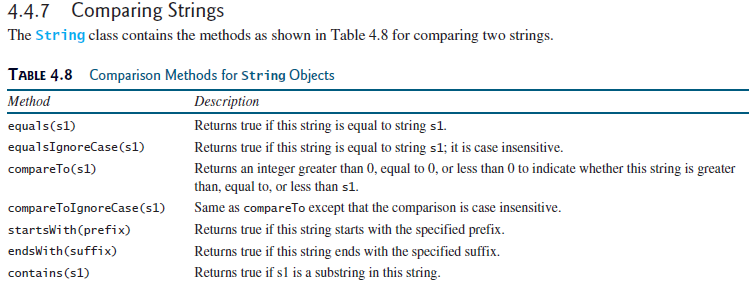
It is the same as doing: String s4 = s1 + s2;

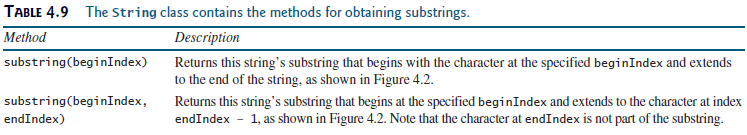
The + can be used in concatenation, as here, and in addition, as in int num = 2 + 2;

In order to be used in concatenation, at least one of the operands has to be a String.

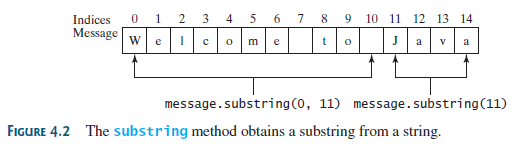
When you are joining two or more words, you have to make sure that you are thinking of the space between the words. For instance, if you are joining the word hi and the word there, you want the new word to be hi there, not hithere.

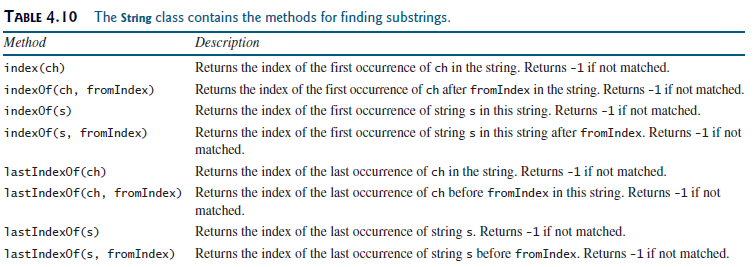


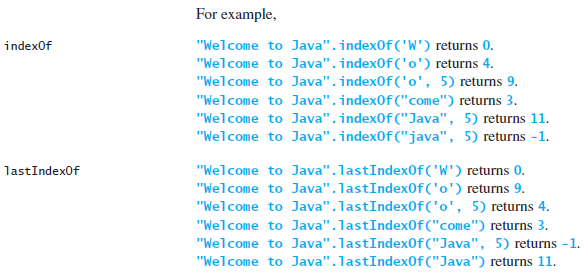




Notice the beginIndex and endIndex values in the substring method; beginIndex values means starting at this spot, the endIndex value means up to – NOT including – that index spot.







# regex

Regular expressions are a way to describe a set of strings based on common characteristics shared by each string in the set. They can be used to search, edit, or manipulate text and data. You must learn a specific syntax to create regular expressions — one that goes beyond the normal syntax of the Java programming language. Regular expressions vary in complexity, but once you understand the basics of how they're constructed, you'll be able to decipher (or create) any regular expression.

We will be using the matches method to find the patterns in Strings. This method will return a boolean value when comparing the String to the pattern.

## Ways we can set up patterns:

|  |  |
| --- | --- |
| **Character Class** | **Description** |
| [abc] | a, b, or c (simple class) |
| [^abc] | Any character except a, b, or c (negation) |
| [a-zA-Z] | a through z or A through Z, inclusive (range) |
| [a-d[m-p]] | a through d, or m through p: [a-dm-p] (union) |
| [a-z&&[def]] | d, e, or f (intersection) |
| [a-z&&[^bc]] | a through z, except for b and c: [ad-z] (subtraction) |
| [a-z&&[^m-p]] | a through z, and not m through p: [a-lq-z](subtraction) |

The regular expression metacharacters work as a short codes.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| . | Any character (may or may not match terminator) |
| \d | Any digits, short of [0-9] |
| \D | Any non-digit, short for [^0-9] |
| \s | Any whitespace character, short for [\t\n\x0B\f\r] |
| \S | Any non-whitespace character, short for [^\s] |
| \w | Any word character, short for [a-zA-Z\_0-9] |
| \W | Any non-word character, short for [^\w] |
| \b | A word boundary |
| \B | A non word boundary |

The quantifiers specify the number of occurrences of a character.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| X? | X occurs once or not at all |
| X+ | X occurs once or more times |
| X\* | X occurs zero or more times |
| X{n} | X occurs n times only |
| X{n,} | X occurs n or more times |
| X{y,z} | X occurs at least y times but less than z times |

Examples:

// this is looking for the pattern of 3 whole numbers, a dash, 2 whole numbers, a dash, 4 whole numbers

System.out.println("440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}"));

// given a string with 4 characters that make up words, a space, and 4 numbers

String stuff = "aaAa 1234";

System.out.println(stuff.matches("\\w{4} \\d{4}"));

System.out.println(stuff.matches("[A-Za-z]{4} [0-9]{4}"));

# StingBuilder & StingBuffer Classes

In general, StringBuilder & StringBuffer classes can be used whenever a String is used.

StringBuilder & StringBuffer are more flexible than String because they can be modified.

* Can insert
* Can append
* Can delete characters
* Can replace characters

StringBuilder is more efficient if it is accessed by just a single task

StringBuffer is synchronized (only one task is allowed to execute methods) = used in concurrent programming – multiple tasks running concurrently = multithreading & parallel

When to use what – a String, a StringBuilder, or a StringBuffer

* String – when the object is not to be changed \*\* it is more efficient (in memory storage)
  + If the string is modified over & over then unused objects are left in memory waiting for the garbage collector
* StringBuilder – when a single program is using the string
  + Are not thread safe
* StringBuffer – when there is multithreading / parallel access to the string

the methods (except setCharAt) do 2 things

1. Changes the content of the string builder/buffer
2. Returns the reference of the string builder/buffer

The principal operations on a StringBuilder that are not available in String are the append() and insert() methods, which are overloaded so as to accept data of any type. Each converts its argument to a string and then appends or inserts the characters of that string to the character sequence in the string builder. The append method always adds these characters at the end of the existing character sequence, while the insert method adds the characters at a specified point.

\*\*\* Internally a string builder is an array of characters

The difference of length & capacity

* Length = the actual size of the string stored in the builder
* Capacity = the current size of the builder
  + The builder's capacity is automatically increased if more characters are added to exceed its capacity
  + The size of the array
    - If the size of the array *– the capacity -* is exceeded, the array is replaced with a new array
      * New array = 2 \* (the previous array size + 1)

|  |  |
| --- | --- |
| **StringBuilder Constructors** | |
| **Constructor** | **Description** |
| StringBuilder() | Creates an empty string builder with a capacity of 16 (16 empty elements). |
| StringBuilder(CharSequence cs) | Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence. |
| StringBuilder(int initCapacity) | Creates an empty string builder with the specified initial capacity. |
| StringBuilder(String s) | Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string. |

For example, the following code

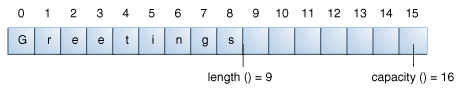
// creates empty builder, capacity 16

StringBuilder sb = new StringBuilder();

// adds 9 character string at beginning

sb.append("Greetings");

will produce a string builder with a length of 9 and a capacity of 16:



The StringBuilder class has some methods related to length and capacity that the String class does not have:

|  |  |
| --- | --- |
| **Length and Capacity Methods** | |
| **Method** | **Description** |
| void setLength(int newLength) | Sets the length of the character sequence. If newLength is less than length(), the last characters in the character sequence are truncated. If newLength is greater than length(), null characters are added at the end of the character sequence. |
| void ensureCapacity(int minCapacity) | Ensures that the capacity is at least equal to the specified minimum. |

A number of operations (for example, append(), insert(), or setLength()) can increase the length of the character sequence in the string builder so that the resultant length() would be greater than the current capacity(). When this happens, the capacity is automatically increased.

**StringBuilder Operations**

The principal operations on a StringBuilder that are not available in String are the append() and insert() methods, which are overloaded so as to accept data of any type. Each converts its argument to a string and then appends or inserts the characters of that string to the character sequence in the string builder. The append method always adds these characters at the end of the existing character sequence, while the insert method adds the characters at a specified point.

Here are a number of the methods of the StringBuilder class.

|  |  |
| --- | --- |
| **Various StringBuilder Methods** | |
| **Method** | **Description** |
| StringBuilder append(boolean b) StringBuilder append(char c) StringBuilder append(char[] str) StringBuilder append(char[] str, int offset, int len) StringBuilder append(double d) StringBuilder append(float f) StringBuilder append(int i) StringBuilder append(long lng) StringBuilder append(Object obj) StringBuilder append(String s) | Appends the argument to this string builder. The data is converted to a string before the append operation takes place. |
| StringBuilder delete(int start, int end) StringBuilder deleteCharAt(int index) | The first method deletes the subsequence from start to end-1 (inclusive) in the StringBuilder's char sequence. The second method deletes the character located at index. |
| StringBuilder insert(int offset, boolean b) StringBuilder insert(int offset, char c) StringBuilder insert(int offset, char[] str) StringBuilder insert(int index, char[] str, int offset, int len) StringBuilder insert(int offset, double d) StringBuilder insert(int offset, float f) StringBuilder insert(int offset, int i) StringBuilder insert(int offset, long lng) StringBuilder insert(int offset, Object obj) StringBuilder insert(int offset, String s) | Inserts the second argument into the string builder. The first integer argument indicates the index before which the data is to be inserted. The data is converted to a string before the insert operation takes place. |
| StringBuilder replace(int start, int end, String s) void setCharAt(int index, char c) | Replaces the specified character(s) in this string builder. |
| StringBuilder reverse() | Reverses the sequence of characters in this string builder. |
| String toString() | Returns a string that contains the character sequence in the builder. |

**Note:** You can use any String method on a StringBuilder object by first converting the string builder to a string with the toString() method of the StringBuilder class. Then convert the string back into a string builder using the StringBuilder(String str) constructor.

Examples:

// capacity and length of StringBuilders

// default StringBuilder instantiated

StringBuilder sb = new StringBuilder();

System.out.println("sb constructed is " + sb);

// sb's capacity is 16 - default capacity size

System.out.println("sb's capacity is " + sb.capacity());

// sb's length is 0 - empty

System.out.println("sb's length is " + sb.length());

Sb.append("Adding to the StringBuilder");